LAB2 : DEEP LEARNING

Neural Network : Second Implementation with a Deep Neural Network

Florian Bertelli, Timothée Babinet, Ramine Hamidi

ABSTRACT

In the following document our aim is to implement a neural network, using numpy, without using a neural network library (like TF, Pytorch). The task will be to predict the digits, with the famous MNIST database, using a Deep Neural Network (with hidden layers) using various activation function (ReLu or tanh)

Data description

- The data are downloaded from the MNIST database, which contains 50 000 images of hand-written digits, each image being associated with the corresponding label (integer). Each image has a resolution of 28 * 28 pixels.

Summary

- This lab contains a full implementation of a neural network, with the training of a linear classifier using : $o = \text{softmax}(Wx + b)$
  * x is an input image that is represented as a vector, each value being the grey intensity of a pixel
  * W and b are the parameters of the classifier
  * softmax transforms the output weight (logits) into probabilities
  * o is a column vector that contains the probability of each category

- Our loss function, that we will try to minimize during the training is :

$$\mathcal{L}(x, gold) = -\log \frac{\exp(x[gold])}{\sum_j \exp(x[j])}$$

- Then it contains a deep neural network (with hidden layers) and with the activation function *tanh* or *Relu*. This will enables us to have a more complex neural network that will be able to model better the MNIST dataset and so achieve a better accuracy.

- To implement this more complex structures we've used the principle of Tensor. Each node is represented by a tensor where you can find the data, the pointer to the precedent tensor to backpropate, the gradient, and the derivative.

Parameters initialisation

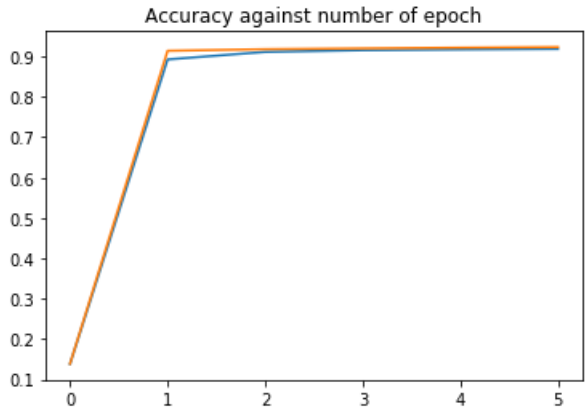
- As we are doing an affine transform we decided to initialise
  - The bias $b_0 = 0$
  - The weights $W \sim [-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}; \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}]$

- If the activation function is *ReLu* we used Kaiming
  - The bias $b_0 = 0$
  - The weights $W \sim [-\frac{\sqrt{6}}{\sqrt{n_j}}; \frac{\sqrt{6}}{\sqrt{n_j}}]$

- If the activation function is *ReLu* we used Goriot
  - The bias $b_0 = 0$
  - The weights $W \sim [-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}; \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}]$

Linear Implementation

- First, we did a linear implementation as in the first lab. Indeed, we just implemented a neural network, where the learning rate is fixed before and not adaptive, which keeps the last W and b.
  With our best implementation, and 5 epochs, we get:
  1. Accuracy on train data : 0.91878
  2. Accuracy on dev data : 0.9234
  3. Accuracy on test data : 0.921



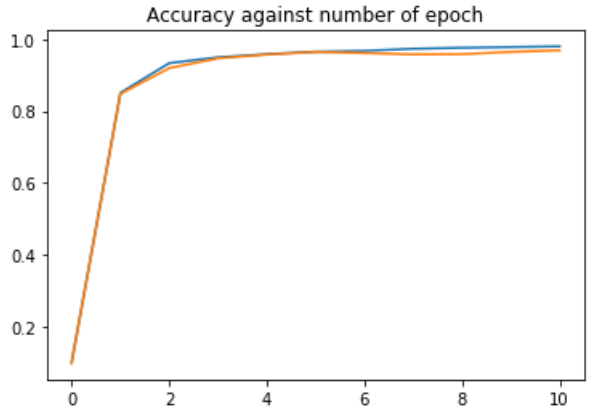
As we can see it is really easy to obtain such results (a simple linear neural network is enough). But the aim of Neural Network is to use deep representations to model new features. This network is quite limited and doesn't manage to learn all the hidden features of the data set, thus to do almost perfect predictions

Deep Neural Network with *ReLu*

- In this part we implement a Deep Neural Network with one (or possibly more hidden layers) to be able

to learn more features. To do that we add one hidden layer with the activation function $ReLu$ ($_tanh = False$)

1. Accuracy on train data : 0.98004
2. Accuracy on dev data : 0.9689
3. Accuracy on test data : 0.9703

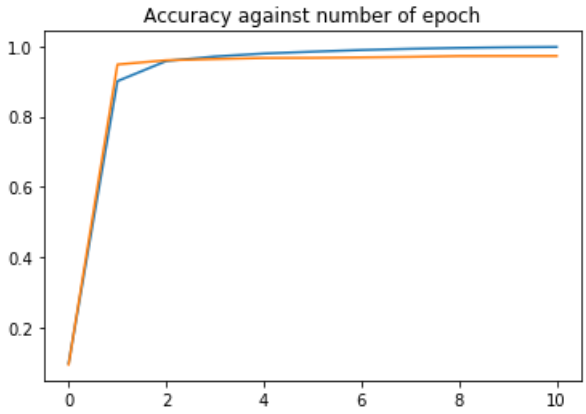


Even if we get a little bit of over fitting (the blue curve corresponding to training increases while the orange one corresponding to evaluating is flat), the results are quite good on test data 97 percent of accuracy with 10 epochs. As we can see this implementation is way more power full than the simple linear one because its complexity permits to model more complex features and relations.

Deep Neural Network with *tanh*

- In this part we implement a Deep Neural Network with one (or possibly more hidden layers) to be able to learn more features. To do that we add one hidden layer with the activation function $tanh$ ($_tanh = True$)

1. Accuracy on train data : 0.99794
2. Accuracy on dev data :0.9723
3. Accuracy on test data : 0.9744



Even if we get a little bit of over fitting (the blue curve corresponding to training increases while the orange one corresponding to evaluating is flat), the results are quite good on test data 97.4 percent of accuracy with 10 epochs. As we can see this implementation is way more power full than the simple linear one because its complexity permits to model more complex features and relations.

Conclusion

To conclude, the "Deep" Neural Networks are more power full than the first Linear Neural Network because they are able to represent more complex model and relations. With just one hidden layer (so a not really deep neural network) we get very good results (97 percent)